Cayenne Guide

Cayenne Guide

License

Table of Contents

List of Tables

Part I. Object Relational Mapping with Cayenne

Chapter 1. Setup

System Requirements

- *Java*: Cayenne runtime framework and CayenneModeler GUI tool are written in 100% Java, and run on any Java-compatible platform. Required JDK version is 1.5 or higher. The last version of Cayenne compatible with JDK 1.4 is 1.2.x/2.0.x; JDK 1.3 - 1.1.x)
- *JDBC Driver:* An appropriate DB-specific JDBC driver is needed to access the database. It can be included in the application or used in web container DataSource configuration.
- *Third-party Libraries:* Cayenne runtime framework has a minimal set of required and a fe more optional dependencies on third-party open source packages. See "Including Cayenne in a Project" chapter for details.

Running CayenneModeler

CayenneModeler GUI tool is intended to work with object relational mapping projects. While you can edit your XML by hand, it is rarely needed, as the Modeler is a pretty advanced tool included in Cayenne distribution. To obtain CayenneModeler, download Cayenne distribution archive from http://cayenne.apache.org/download.html matching the OS you are using. Of course Java needs to be installed on the machine where you are going to run the Modeler.

OS X distribution contains CayenneModeler.app at the root of the distribution disk image.

Windows distribution contains CayenneModeler.exe file in the `bin` directory.

Cross-platform distribution (targeting Linux, but as the name implies, compatible with any OS) contains a runnable CayenneModeler.jar in the `bin` directory. It can be executed either by double-clicking, or if the environment is not configured to execute jars, by running from command-line:

```
java -jar CayenneModeler.jar
```

The Modeler can alo be started from Maven. While it may look like an exotic way to start a GUI application, it has its benefits - no need to download Cayenne distribution, the version of the Modeler always matches the version of the framework, the plugin can find mapping files in the project automatically. So is an attractive option to some developers. Maven option requires a declaration in the POM:

```
<build>
    <plugins>
        <plugin>
            <groupId>org.apache.cayenne.plugins</groupId>
            <artifactId>maven-cayenne-modeler-plugin</artifactId>
            <version>X.Y.Z</version>
        </plugin>
    </plugins>
```

```
</build>
```

And then can be run as

```
mvn cayenne-modeler:run
```

Chapter 2. Cayenne Mapping Structure

Cayenne Project

DataMap

DataNode

DbEntity

ObjEntity

Mapping ObjAttributes to Custom Classes

Embeddable

Procedure

Query

Listeners and Callbacks

Chapter 3. CayenneModeler Application

Working with Mapping Projects

Reverse Engineering Database

Generating Database Schema

Migrations

Generating Java Classes

Modeling Inheritance

Modeling Primary Key Generation Strategy

Part II. Cayenne Framework

Chapter 4. Including Cayenne in a Project

Jar Files and Dependencies

Cayenne distribution contains the following core runtime jars in the distribution `lib` directory:

- *cayenne-server-x.x.jar* - contains full Cayenne runtime (DI, adapters, DB access classes, etc.). Most applications will use only this file.
- cayenne-client-x.x.jar - a subset of cayenne-server.jar trimmed for use on the client in an ROP application.
- Other cayenne-* jars - various Cayenne extensions.

When using `cayenne-server-x.x.jar` you'll need a few third party jars (all included in `lib/third-party` directory of the distribution):

- Apache Velocity Template Engine, version 1.6.x (and all its dependencies bundled with velocity-dep)
- Apache Commons Collections, version 3.2.1
- Apache Commons Logging, version 1.1

Cayenne integrates with various caching, clustering and other frameworks. These optional integrations will require other third-party jars that the users will need to obtain on their own.

Maven Projects

If you are using Maven, you won't have to deal with figuring out the dependencies. You can simply include cayenne-server artifact in your POM:

```
<dependency>
   <groupId>org.apache.cayenne</groupId>
   <artifactId>cayenne-server</artifactId>
   <version>X.Y.Z</version>
</dependency>
```

Additionally Cayenne provides a Maven plugin with a set of goals to perform various project tasks, such as synching generated Java classes with the mapping, described in the following subsection. The full plugin name is `org.apache.cayenne.plugins:maven-cayenne-plugin`.

cgen

`cgen` is a `maven-cayenne-plugin` goal that generates and maintains source (.java) files of persistent objects based on a DataMap. By default, it is bound to the generate-sources phase. If "makePairs" is set to

"true" (which is the recommended default), this task will generate a pair of classes (superclass/subclass) for each ObjEntity in the DataMap. Superclasses should not be changed manually, since they are always overwritten. Subclasses are never overwritten and may be later customized by the user. If "makePairs" is set to "false", a single class will be generated for each ObjEntity.

By creating custom templates, you can use cgen to generate other output (such as web pages, reports, specialized code templates) based on DataMap information.

Table 4.1. cgen required parameters

Name	Type	Description
`map`	File	DataMap XML file which serves as a source of metadata for class generation. E.g. `${project.basedir}/src/main/resources/my.map.xml`
`destDir`	File	Root destination directory for Java classes (ignoring their package names).

Table 4.2. cgen optional parameters

Name	Type	Description
`additionalMaps`	File	A directory that contains additional DataMap XML files that may be needed to resolve cross-DataMap relationships for the the main DataMap, for which class generation occurs.
`client`	boolean	Whether we are generating classes for the client tier in a Remote Object Persistence application. "False" by default.
`embeddableTemplate`	String	Location of a custom Velocity template file for Embeddable class generation. If omitted, default template is used.
`embeddableSuperTemplate`	String	Location of a custom Velocity template file for Embeddable superclass generation. Ignored unless "makepairs" set to "true". If omitted, default template is used.
`encoding`	String	Generated files encoding if different from the default on current platform. Target encoding must be supported by the JVM running the build. Standard encodings supported by Java on all platforms are US-ASCII, ISO-8859-1, UTF-8, UTF-16BE, UTF-16LE, UTF-16. See javadocs for java.nio.charset.Charset for more information.
`excludeEntities`	String	A comma-separated list of ObjEntity patterns (expressed as a perl5 regex) to exclude from template generation. By default none of the DataMap entities are excluded.
`includeEntities`	String	A comma-separated list of ObjEntity patterns (expressed as a perl5 regex) to include from template generation. By default all DataMap entities are included.

Name	Type	Description
makePairs	boolean	If "true" (a recommended default), will generate subclass/ superclass pairs, with all generated code placed in superclass.
mode	String	Specifies class generator iteration target. There are three possible values: "entity" (default), "datamap", "all". "entity" performs one generator iteration for each included ObjEntity, applying either standard to custom entity templates. "datamap" performs a single iteration, applying DataMap templates. "All" is a combination of entity and datamap.
overwrite	boolean	Only has effect when "makePairs" is set to "false". If "overwrite" os "true", will overwrite older versions of generated classes.
superPkg	String	Java package name of generated superclasses. Only has effect if "makepairs" and "usePkgPath" are set to "true" (both are true by default). Defines a common package for all generated Java classes. If omitted, each superclass will be placed in the same package as subclass.
superTemplate	String	Location of a custom Velocity template file for ObjEntity superclass generation. Only has effect if "makepairs" set to "true". If omitted, default template is used.
template	String	Location of a custom Velocity template file for ObjEntity class generation. If omitted, default template is used.
usePkgPath	boolean	If set to "true" (default), a directory tree will be generated in "destDir" corresponding to the class package structure, if set to "false", classes will be generated in "destDir" ignoring their package.

Example - a typical class generatio scenario, where pairs of classes are generated, and superclasses are placed in a separate package:

```
<plugin>
 <groupId>org.apache.cayenne.plugins</groupId>
 <artifactId>maven-cayenne-plugin</artifactId>
 <version>X.Y.Z</version>

 <!--
 There's an intermittent problem when using Maven/cgen in Eclipse with  m2eclipse plugin that
    requires placing "configuration" section at the plugin level, instead of execution
    level.
 -->
 <configuration>
  <map>${project.basedir}/src/main/resources/my.map.xml</map>
  <destDir>${project.basedir}/src/main/java</destDir>
  <superPkg>org.example.model.auto</superPkg>
 </configuration>
```

```
 <executions>
  <execution>
   <goals>
    <goal>cgen</goal>
   </goals>
  </execution>
 </executions>
</plugin>
```

cdbgen

`cdbgen` is a `maven-cayenne-plugin` goal that drops and/or generates tables in a database on Cayenne DataMap. By default, it is bound to the pre-integration-test phase.

Table 4.3. cdbgen required parameters

Name	Type	Description
`map`	File	DataMap XML file which serves as a source of metadata for DB schema generation. E.g. `${project.basedir}/src/main/resources/my.map.xml`
`driver`	String	A class of JDBC driver to use for the target database.
`url`	String	JDBC connection URL of a target database.

Table 4.4. cdbgen optional parameters

Name	Type	Description
`adapter`	String	Java class name implementing org.apache.cayenne.dba.DbAdapter. While this a it is highly recommended to specify correct target adapter.
`createFK`	boolean	Indicates whether cdbgen should create foreign key constraints. Default is "true"
`createPK`	boolean	Indicates whether cdbgen should create Cayenne-specific auto PK objects. Defa
`createTables`	boolean	Indicates whether cdbgen should create new tables. Default is "true".
`dropPK`	boolean	Indicates whether cdbgen should drop Cayenne primary key support objects. De
`dropTables`	boolean	Indicates whether cdbgen should drop the tables before attempting to create new
`password`	String	Database user password.
`username`	String	Database user name.

Example - creating a DB schema on a local HSQLDB database:

```
<plugin>
 <groupId>org.apache.cayenne.plugins</groupId>
 <artifactId>maven-cayenne-plugin</artifactId>
 <version>X.Y.Z</version>

 <executions>
  <execution>
   <configuration>
```

```
    <map>${project.basedir}/src/main/resources/my.map.xml</map>
    <url>jdbc:hsqldb:hsql://localhost/testdb</url>
    <adapter>org.apache.cayenne.dba.hsqldb.HSQLDBAdapter</adapter>
    <driver>org.hsqldb.jdbcDriver</driver>
    <username>sa</username>
   </configuration>
   <goals>
    <goal>cdbgen</goal>
   </goals>
  </execution>
 </executions>
</plugin>
```

cdbimport

`cdbimport` is a `maven-cayenne-plugin` goal that generates a DataMap based on an existing database schema. By default, it is bound to the generate-sources phase. This allows you to generate your DataMap prior to building your project, which may be necessary if you are also using the cgen task.

Table 4.5. cdbimport required parameters

Name	Type	Description
`map`	File	DataMap XML file which is the destination of the schema import. Maybe an existing file. If this file does not exist, it is created when cdbimport is executed. E.g. `${project.basedir}/src/main/resources/my.map.xml`
`driver`	String	A class of JDBC driver to use for the target database.
`url`	String	JDBC connection URL of a target database.

Table 4.6. cdbimport optional parameters

Name	Type	Description
`adapter`	String	Java class name implementing org.apache.cayenne.dba.DbA highly recommended to specify correct target adapter.
`importProcedures`	boolean	Indicates whether stored procedures should be imported from
`meaningfulPk`	boolean	Indicates whether primary keys should be mapped as attribut
`namingStrategy`	String	The naming strategy used for mapping database names to ob `org.apache.cayenne.map.naming.SmartNaming`
`overwriteExisting`	boolean	Indicates whether existing DB and object entities should be o CayenneModeler. Default is "true".
`password`	String	Database user password.
`procedurePattern`	String	Pattern to match stored procedure names against for import. importProcedures is true.
`schemaName`	String	Database schema to import tables/stored procedures from.

Name	Type	Description
`tablePattern`	String	Pattern to match table names against for import. Default is to match
`username`	String	Database user name.

Example - loading a DB schema from a local HSQLDB database (essentially a reverse operation compared to the cdbgen example above) :

```
<plugin>
 <groupId>org.apache.cayenne.plugins</groupId>
 <artifactId>maven-cayenne-plugin</artifactId>
 <version>X.Y.Z</version>

 <executions>
  <execution>
   <configuration>
    <map>${project.basedir}/src/main/resources/my.map.xml</map>
    <url>jdbc:hsqldb:hsql://localhost/testdb</url>
    <adapter>org.apache.cayenne.dba.hsqldb.HSQLDBAdapter</adapter>
    <driver>org.hsqldb.jdbcDriver</driver>
    <username>sa</username>
   </configuration>
   <goals>
    <goal>cdbimport</goal>
   </goals>
  </execution>
 </executions>
</plugin>
```

Ant Projects

cgen

cdbgen

cdbimport

cdataport

Chapter 5. Starting Cayenne

Starting and Stopping ServerRuntime

In runtime Cayenne is accessed via `org.apache.cayenne.configuration.server.ServerRuntime`. ServerRuntime is created simply by calling a constructor:

```
ServerRuntime runtime =
 new ServerRuntime("com/example/cayenne-project.xml");
```

The parameter you pass to the constructor is a location of the main project file. Location is a '/'-separated path (same path separator is used on UNIX and Windows) that is resolved relative to the application classpath. The project file can be placed in the root package or in a subpackage (e.g. in the code above it is in "com/example" subpackage).

ServerRuntime encapsulates a single Cayenne stack. Most applications will just have one ServerRuntime using it to create as many ObjectContexts as needed, access the Dependency Injection (DI) container and work with other Cayenne features. Internally ServerRuntime is just a thin wrapper around the DI container. Detailed features of the container are discussed in "Customizing Cayenne Runtime" chapter. Here we'll just show an example of how an application might replace a default implementation of a built-in Cayenne service (in this case - QueryCache) with a different class:

```
public class MyExtensionsModule implements Module {
 public void configure(Binder binder) {
  binder.bind(QueryCache.class).to(EhCacheQueryCache.class);
 }
}

Module extensions = new MyExtensionsModule();
ServerRuntime runtime =
 new ServerRuntime("com/example/cayenne-project.xml", extensions);
```

It is a good idea to shut down the runtime when it is no longer needed, usually before the application itself is shutdown:

```
runtime.shutdown();
```

When a runtime object has the same scope as the application, this may not be always necessary, however in some cases it is essential, and is generally considered a good practice. E.g. in a web container hot redeploy of a webapp will cause resource leaks and eventual OutOfMemoryError if the application fails to shutdown CayenneRuntime.

Merging Multiple Projects

ServerRuntime requires at least one mapping project to run. But it can also take multiple projects and merge them together in a single configuration. This way different parts of a database can be mapped independenlty

from each other (even by different software providers), and combined in runtime when assembling an application. Doing it is as easy as passing multiple project locations to ServerRuntime constructor:

```
ServerRuntime runtime =
 new ServerRuntime(new String[] {
  "com/example/cayenne-project.xml",
  "org/foo/cayenne-library1.xml",
  "org/foo/cayenne-library2.xml"
 }
);
```

When the projects are merged, the following rules are applied:

- The order of projects matters during merge. If there are two conflicting metadata objects belonging to two projects, an object from the *last* project takes precedence over the object from the first one. This makes possible to override pieces of metadata. This is also similar to how DI modules are merged in Cayenne.
- Runtime DataDomain name is set to the name of the last project in the list.
- Runtime DataDomain properties are the same as the properties of the last project in the list. I.e. *properties are not merged* to avoid invalid combinations and unexpected runtime behavior.
- If there are two or more DataMaps with the same name, only one DataMap is used in the merged project, the rest are discarded. Same precedence rules apply - DataMap from the project with the highest index in the project list overrides all other DataMaps with the same name.
- If there are two or more DataNodes with the same name, only one DataNodes is used in the merged project, the rest are discarded. DataNode coming from project with the highest index in the project list is chosen per precedence rule above.
- There is a notion of "default" DataNode. After the merge if any DataMaps are not explicitly linked to DataNodes, their queries will be executed via a default DataNode. This makes it possible to build mapping "libraries" that are only associated with a specific database in runtime. If there's only one DataNode in the merged project, it will be automatically chosen as default. A possible way to explicitly designate a specific node as default is to override `DataDomainProvider.createAndInitDataDomain()`.

Web Applications

Chapter 6. Persistent Objects and ObjectContext

ObjectContext

ObjectContext is an interface that users normally work with to access the database. It provides the API to execute database operations and to manage persistent objects. A context is obtained from the ServerRuntime:

```
ObjectContext context = runtime.getContext();
```

The call above creates a new instance of ObjectContext that can access the database via this runtime. ObjectContext is a single "work area" in Cayenne, storing persistent objects. ObjectContext guarantees that for each database row with a unique ID it will contain at most one instance of an object, thus ensuring object graph consistency between multiple selects (a feature called "uniquing"). At the same time different ObjectContexts will have independent copies of objects for each unique database row. This allows users to isolate object changes from one another by using separate ObjectContexts.

These properties directly affect the strategies for scoping and sharing (or not sharing) ObjectContexts. Contexts that are only used to fetch objects from the database and whose objects are never modified by the application can be shared between mutliple users (and multiple threads). Contexts that store modified objects should be accessed only by a single user (e.g. a web application user might reuse a context instance between multiple web requests in the same HttpSession, thus carrying uncommitted changes to objects from request to request, until he decides to commit or rollback them). Even for a single user it might make sense to use mutliple ObjectContexts (e.g. request-scoped contexts to allow concurrent requests from the browser that change and commit objects independently).

ObjectContext is serializable and does not permanently hold to any of the application resources. So it does not have to be closed. If the context is not used anymore, it should simply be allowed to go out of scope and get garbage collected, just like any other Java object.

Persistent Object and its Lifecycle

Cayenne can persist Java objects that implement `org.apache.cayenne.Persistent` interface. Generally persistent objects are created from the model via class generation as described above, so users do not have to worry about implementation details.

Persistent interface provides access to 3 persistence-related properties - objectId, persistenceState and objectContext. All 3 are initialized by Cayenne runtime framework. Application code should not attempt to change them them. However it is allowed to read them, which provides valuable runtime information. E.g. ObjectId can be used for quick equality check of 2 objects, knowing persistence state would allow highlighting changed objects, etc.

Each persistent object belongs to a single ObjectContext, and can be in one of the following persistence states (as defined in `org.apache.cayenne.PersistenceState`):

Table 6.1. Persistence States

TRANSIENT	The object is not registered with an ObjectContext and will not be persisted.
NEW	The object is freshly registered in an ObjectContext, but has not been saved to the database yet and there is no matching database row.
COMMITTED	The object is registered in an ObjectContext, there is a row in the database corresponding to this object, and the object state corresponds to the last known state of the matching database row.
MODIFIED	The object is registered in an ObjectContext, there is a row in the database corresponding to this object, but the object in-memory state has diverged from the last known state of the matching database row.
HOLLOW	The object is registered in an ObjectContext, there is a row in the database corresponding to this object, but the object state is unknown. Whenever an application tries to access a property of such object, Cayenne attempts reading its values from the database and "inflate" the object, turning it to COMMITED.
DELETED	The object is registered in an ObjectContext and has been marked for deletion in-memory. The corresponding row in the database will get deleted upon ObjectContext commit, and the object state will be turned into TRANSIENT.

ObjectContext Persistence API

One of the first things users usually want to do with an ObjectContext is to select some objects from an existing database. This is done by calling "*performQuery*" method:

```
SelectQuery query = new SelectQuery(Artist.class);
List<Artist> artists = context.performQuery(query);
```

We'll discuss queries in some detail in the following chapters. The example above is self-explanatory - we create a SelectQuery that matches all Artist objects present in the database, and then call "performQuery", getting a list of Artist objects.

In some cases queries can be quite complex, returning multiple result sets, and even updating the database. For such queries ObjectContext provides "*performGenericQuery*"method. While not nearly as common as "performQuery", it is nevertheless important in some situations. E.g.:

```
Collection<Query> queries = ... // some queries
QueryChain query = new QueryChain(queries);

QueryResponse response = context.performGenericQuery(query);
```

The "*newObject*" method call creates a new persistent object setting its state to NEW:

```
Artist artist = context.newObject(Artist.class);
artist.setName("Picasso");
```

Once a new object is created, its properties can be modified by the application in memory without affecting the database. To ensure the object is saved to the database, application must call "*commitChanges*":

```
context.commitChanges();
```

In our case "commitChanges" commits just this one artist object, but in fact it commits all in-memory changes to all objects registered in this ObjectContext (it just happens that we didn't have any more objects to commit). I.e. anything that has changed since the previous commit or rollback (or since the context creation if there were no previous commits or rollbacks). Commit internally generates a minimal set of SQL statements to synchronize the database with the in-memory state of all changed objects and sends them to DB in a single transaction.

Cayenne Helper Class

ObjectContext Nesting

Generic Persistent Objects

Transactions

Chapter 7. Expressions

Expressions Overview

Path Expressions

Creating Expressions from Strings

Creating Expressions with ExpressionFactory

Evaluating Expressions in Memory

Chapter 8. Queries

SelectQuery

EJBQLQuery

SQLTemplateQuery

ProcedureQuery

NamedQuery

Custom Queries

Chapter 9. Lifecycle Events

Types of Lifecycle Events

Lifecycle Callbacks and Listeners

Callback and Listener Methods Semantics

Registering Callbacks and Listeners

Combining Listeners with DataChannelFilters

Chapter 10. Performance Tuning

Prefetching

Prefetching is a technique that allows to bring back in one query not only the queried objects, but also objects related to them. In other words it is a controlled eager relationship resolving mechanism. Prefetching is discussed in the "Performance Tuning" chapter, as it is a powerful performance optimization method. Another common application of prefetching is for refreshing stale object relationships.

Prefetching example:

```
SelectQuery query = new SelectQuery(Artist.class);

// this instructs Cayenne to prefetch one of Artist's relationships
query.addPrefetch("paintings");

// query is expecuted as usual, but the resulting Artists will have
// their paintings "inflated"
List<Artist> artists = context.performQuery(query);
```

All types of relationships can be preftetched - to-one, to-many, flattened.

A prefetch can span multiple relationships:

```
 query.addPrefetch("paintings.gallery");
```

A query can have multiple prefetches:

```
query.addPrefetch("paintings");
query.addPrefetch("paintings.gallery");
```

If a query is fetching DataRows, all "disjoint" prefetches are ignored, only "joint" prefetches are executed (see prefetching semantics discussion below for what disjoint and joint prefetches mean).

Prefetching Semantics

Prefetching semantics defines a strategy to prefetch relationships. Depending on it, Cayenne would generate different types of queries. The end result is the same - query root objects with related objects fully resolved. However semantics can affect preformance, in some cases significantly. There are 3 types of prefetch semantics, all defined as constants in org.apache.cayenne.query.PrefetchTreeNode:

```
PrefetchTreeNode.JOINT_PREFETCH_SEMANTICS
PrefetchTreeNode.DISJOINT_PREFETCH_SEMANTICS
PrefetchTreeNode.DISJOINT_BY_ID_PREFETCH_SEMANTICS
```

Each query has a default prefetch semantics, so generally users do not have to worry about changing it, except when performance is a concern, or a few special cases when a default sematics can't produce the correct result. SelectQuery uses DISJOINT_PREFETCH_SEMANTICS by default. Semantics can be changed as follows:

```
SelectQuery query = new SelectQuery(Artist.class);
```

```
query.addPrefetch("paintings").setSemantics(
                PrefetchTreeNode.JOINT_PREFETCH_SEMANTICS);
```

There's no limitation on mixing different types of semantics in the same SelectQuery. Multiple prefetches each can have its own semantics.

SQLTemplate and ProcedureQuery are both using JOINT_PREFETCH_SEMANTICS and it can not be changed due to the nature of these two queries.

Disjoint Prefetching Semantics

This semantics (only applicable to SelectQuery) results in Cayenne generatiing one SQL statement for the main objects, and a separate statement for each prefetch path (hence "disjoint" - related objects are not fetched with the main query). Each additional SQL statement uses a qualifier of the main query plus a set of joins traversing the preftech path between the main and related entity.

This strategy has an advantage of efficient JVM memory use, and faster overall result processing by Cayenne, but it requires (1+N) SQL statements to be executed, where N is the number of prefetched relationships.

Disjoint-by-ID Prefetching Semantics

This is a variation of disjoint prefetch where related objects are matched against a set of IDs derived from the fetched main objects (or intermediate objects in a multi-step prefetch). Cayenne limits the size of the generated WHERE clause, as most DBs can't parse arbitrary large SQL. So prefetch queries are broken into smaller queries. The size of is controlled by the DI property Constants.SERVER_MAX_ID_QUALIFIER_SIZE_PROPERTY (the default number of conditions in the generated WHERE clause is 10000). Cayenne will generate (1 + N * M) SQL statements for each query using disjoint-by-ID prefetches, where N is the number of relationships to prefetch, and M is the number of queries for a given prefetch that is dependent on the number of objects in the result (ideally M = 1).

The advantage of this type of prefetch is that matching database rows by ID may be much faster than matching the qualifier of the original query. Moreover this is **the only type of prefetch** that can handle SelectQueries with **fetch limit**. Both joint and regular disjoint prefetches may produce invalid results or generate inefficient fetch-the-entire table SQL when fetch limit is in effect.

The disadvantage is that query SQL can get unwieldy for large result sets, as each object will have to have its own condition in the WHERE clause of the generated SQL.

Joint Prefetching Semantics

Joint senantics results in a single SQL statement for root objects and any number of jointly prefetched paths. Cayenne processes in memory a cartesian product of the entities involved, converting it to an object tree. It uses OUTER joins to connect prefetched entities.

Joint is the most efficient prefetch type of the three as far as generated SQL goes. There's always just 1 SQL query generated. Its downsides are the potentially increased amount of data that needs to get across the network between the application server and the database, and more data processing that needs to be done on the Cayenne side.

Data Rows

Iterated Queries

Paginated Queries

Caching and Fresh Data

Object Caching

Query Result Caching

Turning off Synchronization of ObjectContexts

Chapter 11. Customizing Cayenne Runtime

Dependency Injection Container

Cayenne runtime is built around a small powerful dependency injection (DI) container. Just like other popular DI technologies, such as Spring or Guice, Cayenne DI container manages sets of interdependent objects and allows users to configure them. These objects are regular Java objects. We are calling them "services" in this document to distinguish from all other objects that are not configured in the container and are not managed. DI container is responsible for service instantiation, injecting correct dependencies, maintaining service instances scope, and dispatching scope events to services.

The services are configured in special Java classes called "modules". Each module defines binding of service interfaces to implementation instances, implementation types or providers of implementation instances. There are no XML configuration files, and all the bindings are type-safe. The container supports injection into instance variables and constructor parameters based on the `@Inject` annotation. This mechanism is very close to Google Guice.

The discussion later in this chapter demonstrates a standalone DI container. But keep in mind that Cayenne already has a built-in Injector, and a set of default modules. A Cayenne user would normally only use the API below to write custom extension modules that will be loaded in that existing container when creating ServerRuntime. See "Starting and Stopping ServerRuntime" chapter for an example of passing an extension module to Cayenne.

Cayenne DI probably has ~80% of the features expected in a DI container and has no dependency on the rest of Cayenne, so in theory can be used as an application-wide DI engine. But it's primary purpose is still to serve Cayenne. Hence there are no plans to expand it beyond Cayenne needs. It is an ideal "embedded" DI that does not interfere with Spring, Guice or any other such framework present elsewhere in the application.

DI Bindings API

To have a working DI container, we need three things: service interfaces and classes, a module that describes service bindings, a container that loads the module, and resolves the depedencies. Let's start with service interfaces and classes:

```
public interface Service1 {
 public String getString();
}

public interface Service2 {
 public int getInt();
}
```

A service implementation using instance variable injection:

```
public class Service1Impl implements Service1 {
 @Inject
 private Service2 service2;

 public String getString() {
  return service2.getInt() + "_Service1Impl";
 }
}
```

Same thing, but using constructor injection:

```
public class Service1Impl implements Service1 {

 private Service2 service2;

 public Service1Impl(@Inject Service2 service2) {
  this.service2 = service2;
 }

 public String getString() {
  return service2.getInt() + "_Service1Impl";
 }
}
```

```
public class Service2Impl implements Service2 {
 private int i;

 public int getInt() {
  return i++;
 }
}
```

Now let's create a module implementing `org.apache.cayenne.tutorial.di.Module` interface that will contain DI configuration. A module binds service objects to keys that are reference. Binder provided by container implements fluent API to connect the key to implementation, and to configure various binding options (the options, such as scope, are demonstrated later in this chapter). The simplest form of a key is a Java Class object representing service interface. Here is a module that binds Service1 and Service2 to corresponding default implementations:

```
public class Module1 implements Module {

 public void configure(Binder binder) {
  binder.bind(Service1.class).to(Service1Impl.class);
  binder.bind(Service2.class).to(Service2Impl.class);
 }
}
```

Once we have at least one module, we can create a DI container. `org.apache.cayenne.di.Injector` is the container class in Cayenne:

```
Injector injector = DIBootstrap.createInjector(new Module1());
```

Now that we have created the container, we can obtain services from it and call their methods:

```
Service1 s1 = injector.getInstance(Service1.class);
for (int i = 0; i < 5; i++) {
 System.out.println("S1 String: " + s1.getString());
```

```
}
```

This outputs the following lines, demonstrating that s1 was Service1Impl and Service2 injected into it was Service2Impl:

```
0_Service1Impl
1_Service1Impl
2_Service1Impl
3_Service1Impl
4_Service1Impl
```

There are more flavors of bindings:

```
// binding to instance - allowing user to create and configure instance
// inside the module class
binder.bind(Service2.class).toInstance(new Service2Impl());

// binding to provider - delegating instance creation to a special
// provider class
binder.bind(Service1.class).toProvider(Service1Provider.class);

// binding to provider instance
binder.bind(Service1.class).toProviderInstance(new Service1Provider());

// multiple bindings of the same type using Key
// injection can reference the key name in annotation:
// @Inject("i1")
// private Service2 service2;
binder.bind(Key.get(Service2.class, "i1")).to(Service2Impl.class);
binder.bind(Key.get(Service2.class, "i2")).to(Service2Impl.class);
```

Another types of confiuguration that can be bound in the container are lists and maps. They will be discussed in the following chapters.

Service Lifecycle

An important feature of the Cayenne DI container is instance *scope*. The default scope (implicitly used in all examples above) is "singleton", meaning that a binding would result in creation of only one service instance, that will be repeatedly returned from `Injector.getInstance(..)`, as well as injected into classes that declare it as a dependency.

Singleton scope dispatches a "BeforeScopeEnd" event to interested services. This event occurs before the scope is shutdown, i.e. when `Injector.shutdown()` is called. Note that the built-in Cayenne injector is shutdown behind the scenes when `ServerRuntime.shutdown()` is invoked. Services may register as listeners for this event by annotating a no-argument method with `@BeforeScopeEnd` annotation. Such method should be implemented if a service needs to clean up some resources, stop threads, etc.

Another useful scope is "no scope", meaning that every time a container is asked to provide a service instance for a given key, a new instance will be created and returned:

```
binder.bind(Service2.class).to(Service2Impl.class).withoutScope();
```

Users can also create their own scopes, e.g. a web application request scope or a session scope. Most often than not custom scopes can be created as instances of

`org.apache.cayenne.di.spi.DefaultScope` with startup and shutdown managed by the application (e.g. singleton scope is a DefaultScope managed by the Injector) .

Overriding Services

Cayenne DI allows to override services already definied in the current module, or more commonly - some other module in the the same container. Actually there's no special API to override a service, you'd just bind the service key again with a new implementation or provider. The last binding for a key takes precedence. This means that the order of modules is important when configuring a container. The built-in Cayenne injector ensures that Cayenne standard modules are loaded first, followed by optional user extension modules. This way the application can override the standard services in Cayenne.

Customization Strategies

The previous section discussed how Cayenne DI works in general terms. Since Cayenne users will mostly be dealing with an existing Injector provided by ServerRuntime, it is important to understand how to build custom extensions to a preconfigured container. As shown in "Starting and Stopping ServerRuntime" chapter, custom extensions are done by writing an aplication DI module (or multiple modules) that configures service overrides. This section shows all the configuration possibilities in detail, including changing properties of the existing services, contributing services to standard service lists and maps, and overriding service implementations. All the code examples later in this section are assumed to be placed in an application module "configure" method:

```
public class MyExtensionsModule implements Module {
 public void configure(Binder binder) {
  // customizations go here...
 }
}
```

```
Module extensions = new MyExtensionsModule();
ServerRuntime runtime =
 new ServerRuntime("com/example/cayenne-mydomain.xml", extensions);
```

Changing Properties of Existing Services

Many built-in Cayenne services change their behavior based on a value of some environment property. A user may change Cayenne behavior without even knowing which services are responsible for it, but setting a specific value of a known property. Supported property names are listed in "Appendix A".

There are two ways to set service properties. The most obvious one is to pass it to the JVM with -D flag on startup. E.g.

```
java -Dorg.apache.cayenne.sync_contexts=false ...
```

A second one is to contribute a property to `org.apache.cayenne.configuration.DefaultRuntimeProperties.properties`

map (see the next section on how to do that). This map contains the default property values and can accept application-specific values, overrding the defaults.

Note that if a property value is a name of a Java class, when this Java class is instantiated by Cayenne, the container performs injection of instance variables. So even the dynamically specified Java classes can use @Inject annotation to get a hold of other Cayenne services.

If the same property is specified both in the command line and in the properties map, the command-line value takes precedence. The map value will be ignored. This way Cayenne runtime can be reconfigured during deployment.

Contributing to Service Collections

Cayenne can be extended by adding custom objects to named maps or lists bound in DI. We are calling these lists/maps "service collections". A service collection allows things like appending a custom strategy to a list of built-in strategies. E.g. an application that needs to install a custom DbAdapter for some database type may contribute an instance of custom DbAdapterDetector to a `org.apache.cayenne.configuration.server.DefaultDbAdapterFactory.detectors` list:

```
public class MyDbAdapterDetector implements DbAdapterDetector {
 public DbAdapter createAdapter(DatabaseMetaData md) throws SQLException {
  // check if we support this database and retun custom adapter
  ...
 }
}
```

```
// since build-in list for this key is a singleton, repeated
// calls to 'bindList' will return the same instance
binder.bindList(DefaultDbAdapterFactory.DETECTORS_LIST)
       .add(MyDbAdapterDetector.class);
```

Maps are customized using a similar "`bindMap`" method.

The names of built-in collections are listed in "Appendix B".

Alternative Service Implementations

As mentioned above, custom modules are loaded by ServerRuntime after the built-in modules. So it is easy to redefine a built-in service in Cayenne by rebinding desired implementations or providers. To do that, first we need to know what those services to redefine are. While we describe some of them in the following sections, the best way to get a full list is to check the source code of the Cayenne version you are using and namely look in `org.apache.cayenne.configuration.server.ServerModule` - the main built-in module in Cayenne.

Now an example of overriding `QueryCache` service. The default implementation of this service is provided by `MapQueryCacheProvider`. But if we want to use `EhCacheQueryCache` (a Cayenne wrapper for the EhCache framework), we can define it like this:

```
binder.bind(QueryCache.class).to(EhCacheQueryCache.class);
```

Noteworthy Built-in Services

JdbcEventLogger

`org.apache.cayenne.log.JdbcEventLogger` is the service that defines logging API for Cayenne internals. It provides facilities for logging queries, commits, transactions, etc. The default implementation is `org.apache.cayenne.log.CommonsJdbcEventLogger` that performs logging via commons-logging library. Cayenne library includes another potentially useful logger - `org.apache.cayenne.log.FormattedCommonsJdbcEventLogger` that produces formatted multiline SQL output that can be easier to read.

DataSourceFactory

DataChannelFilter

QueryCache

ExtendedTypes

Part III. Cayenne Framework - Remote Object Persistence

Chapter 12. Introduction to ROP

What is ROP

Main Features

Chapter 13. ROP Setup

System Requirements

Jar Files and Dependencies

Chapter 14. Implementing ROP Server

Chapter 15. Implementing ROP Client

Chapter 16. ROP Deployment

Deploying ROP Server

Deploying ROP Client

Security

Chapter 17. Current Limitations

Appendix A. Configuration Properties

Note that the property names below are defined as constants in `org.apache.cayenne.configuration.Constants` interface.

Table A.1. Configuration Properties Recognized by ServerRuntime and/or ClientRuntime

Property	Possible Values	Default Value
`cayenne.jdbc.driver[.domain_name.node_name]` - defines a JDBC driver class to use when creating a DataSource. If domain name and optionally - node name are specified, the setting overrides DataSource info just for this domain/node. Otherwise the override is applied to all domains/nodes in the system.		none, project DataNode configuration is used
`cayenne.jdbc.url[.domain_name.node_name]` - defines a DB URL to use when creating a DataSource. If domain name and optionally - node name are specified, the setting overrides DataSource info just for this domain/node. Otherwise the override is applied to all domains/nodes in the system.		none, project DataNode configuration is used
`cayenne.jdbc.username[.domain_name.node_name]` - defines a DB user name to use when creating a DataSource. If domain name and optionally - node name are specified, the setting overrides DataSource info just for this domain/node. Otherwise the override is applied to all domains/nodes in the system.		none, project DataNode configuration is used
`cayenne.jdbc.password[.domain_name.node_name]` - defines a DB password to use when creating a DataSource. If domain name and optionally - node name are specified, the setting overrides DataSource info just for this domain/node. Otherwise the override is applied to all domains/nodes in the system		none, project DataNode configuration is used
`cayenne.jdbc.min_connections[.domain_name.node_name]` - defines the DB connection pool minimal size. If domain name and optionally - node name are specified, the setting overrides DataSource info just for this domain/node. Otherwise the override is applied to all domains/nodes in the system		none, project DataNode configuration is used
`cayenne.jdbc.max_connections[.domain_name.node_name]` - defines the DB connection pool maximum size. If domain name and optionally - node name are specified, the setting overrides DataSource info just for this domain/node. Otherwise the override is applied to all domains/nodes in the system		none, project DataNode configuration is used
`cayenne.querycache.size` - An integer defining the maximum number of entries in the query cache. Note that not all QueryCache providers may respect this property. MapQueryCache uses it, but the rest would use alternative configuration methods.	any positive int value	2000

Property	Possible Values	Default Value
`cayenne.server.contexts_sync_strategy` - defines whether peer ObjectContexts should receive snapshot events after commits from other contexts. If true (default), the contexts would automatically synchronize their state with peers.	true, false	true
`cayenne.server.object_retain_strategy` - defines fetched objects retain strategy for ObjectContexts. When weak or soft strategy is used, objects retained by ObjectContext that have no local changes can potetially get garbage collected when JVM feels like doing it.	weak, soft, hard	weak
`cayenne.server.max_id_qualifier_size` - defines a maximum number of ID qualifiers in the WHERE clause of queries that are generated for paginated queries and for DISJOINT_BY_ID prefetch processing. This is needed to avoid hitting WHERE clause size limitations and memory usage efficiency.	any positive int	10000
`cayenne.rop.service_url` - defines the URL of the ROP server		
`cayenne.rop.service_username` - defines the user name for an ROP client to login to an ROP server.		
`cayenne.rop.service_password` - defines the password for an ROP client to login to an ROP server.		
`cayenne.rop.shared_session_name`- defines the name of the shared session that an ROP client wants to join on an ROP server. If omitted, a dedicated session is created.		
`cayenne.rop.service.timeout` - a value in milliseconds for the ROP client-server connection read operation timeout	any positive long value	
`cayenne.rop.channel_events` - defines whether client-side DataChannel should dispatch events to child ObjectContexts. If set to true, ObjectContexts will receive commit events and merge changes committed by peer contexts that passed through the common client DataChannel.	true, false	false
`cayenne.rop.context_change_events`- defines whether object property changes in the client context result in firing events. Client UI components can listen to these events and update the UI. Disabled by default.	true, false	false
`cayenne.rop.context_lifecycle_events` - defines whether object commit and rollback operations in the client context result in firing events. Client UI components can listen to these events and update the UI. Disabled by default.	true,false	false

Property	Possible Values	Default Value
`cayenne.server.rop_event_bridge_factory` - defines the name of the org.apache.cayenne.event.EventBridgeFactory that is passed from the ROP server to the client. I.e. server DI would provide a name of the factory, passing this name to the client via the wire. The client would instantiate it to receive events from the server. Note that this property is stored in "cayenne.server.rop_event_bridge_properties" map, not in the main "cayenne.properties".		

Appendix B. Service Collections

Note that the collection keys below are defined as constants in `org.apache.cayenne.configuration.Constants` interface.

Table B.1. Service Collection Keys Present in ServerRuntime and/or ClientRuntime

`cayenne.properties` - Map<String,String> of properties used by built-in Cayenne services. The keys in this map are the property names from the table in Appendix A. Separate copies of this map exist on the server and ROP client.

`cayenne.server.adapter_detectors` - List<DbAdapterDetector> that contains objects that can discover the type of current database and install the correct DbAdapter in runtime.

`cayenne.server.domain_filters` - List<DataChannelFilter> storing DataDomain filters.

`cayenne.server.project_locations` - List<String> storing locations of the one of more project configuration files.

`cayenne.server.default_types` - List<ExtendedType> storing default adapter-agnostic ExtendedTypes. Default ExtendedTypes can be overridden / extended by DB-specific DbAdapters as well as by user-provided types configured in another colltecion (see "cayenne.server.user_types").

`cayenne.server.user_types` - List<ExtendedType> storing a user-provided ExtendedTypes. This collection will be merged into a full list of ExtendedTypes and would override any ExtendedTypes defined in a default list, or by a DbAdapter.

`cayenne.server.type_factories` - List<ExtendedTypeFactory> storing default and user-provided ExtendedTypeFactories. ExtendedTypeFactory allows to define ExtendedTypes dynamically for the whole group of Java classes. E.g. Cayenne supplies a factory to map all Enums regardless of their type.

`cayenne.server.rop_event_bridge_properties` - Map<String, String> storing event bridge properties passed to the ROP client on bootstrap. This means that the map is configured by server DI, and passed to the client via the wire. The properties in this map are specific to EventBridgeFactory implementation (e.g JMS or XMPP connection prameters). One common property is "cayenne.server.rop_event_bridge_factory" that defines the type of the factory.